Node.js Stream - 基础篇

stream 提供了以下四种类型的流：

```
var stream = require('stream')
```

```
var Readable = stream.Readablevar Writable = stream.Writablevar Duplex = stream.Duplexvar Transform = stream.Transform
```

使用 Stream 可实现数据的流式处理，如：

```
var fs = require('fs')// `fs.createReadStream`创建一个`Readable`对象以读取`bigFile`的内容，并输出到标准输出// 如果使用`fs.readFile`则可能由于文件过大而失败fs.createReadStream(bigFile).pipe(process.stdout)
```

Readable

创建可读流。

实例：流式消耗迭代器中的数据。

```
'use strict'const Readable = require('stream').Readable
class ToReadable extends Readable {
  constructor(iterable) {
    super()
    this.iterator = new function *() {
      yield * iterable
    }
  }

  // 子类需要实现该方法
  // 这是生产数据的逻辑
  _read() {
```

```
    const res = this.iterator.next()
    if (res.done) {
      // 数据源已枯竭，调用`push(null)`通知流
      this.push(null)
    } else {
      // 通过`push`方法将数据添加到流中
      this.push(res.value + '\n')
    }
  }
}
module.exports = ToReadable
```

实际使用时，new ToReadable(iterable)会返回一个可读流，下游可以流式的消耗迭代器中的数据。

```
const iterable = function *(limit) {
  while (limit--) {
    yield Math.random()
  }
}(1e10)

const readable = new ToReadable(iterable)

// 监听`data`事件，一次获取一个数据 readable.on('data', data =>
process.stdout.write(data))

// 所有数据均已读完 readable.on('end', () => process.stdout.write('DONE'))
```

执行上述代码，将会有 100 亿个随机数源源不断的写进标准输出流。

创建可读流时，需要继承 Readable，并实现_read 方法。

- _read 方法是从底层系统读取具体数据的逻辑，即生产数据的逻辑。
- 在_read 方法中，通过调用 push(data)将数据放入可读流中供下游消耗。

- 在_read 方法中，可以同步地调用 push(data)，也可以异步地调用。
- 当全部数据都生产出来后，**必须**调用 push(null)来结束可读流。
- 流一旦结束，便不能再调用 push(data)添加数据。

可以通过监听 data 事件的方式消耗可读流。

- 在首次监听其 data 事件后，readable 便会持续不断的调用_read()，通过触发 data 事件将数据输出。
- 第一次 data 事件会在下一个 tick 中触发，所以，可以安全地将数据输出前的逻辑放在事件监听后（同一个 tick 中）。
- 当数据全部被消耗时，会触发 end 事件。

上面的例子中，process.stdout 代表标准输出流，实际是一个可写流。下小节中介绍可写流的用法。

Writable

创建可写流。

前面通过继承的方式去创建一类可读流，这种方法也适用于创建一类可写流，只是需要实现的是_write(data, enc, next)方法，而不是_read()方法。

有些简单的情况下不需要创建一类流，而只是一个流对象，可以用如下方式去做：

```
'use strict'const Writable = require('stream').Writable

const writable = Writable()// 实现`_write`方法// 这是将数据写入底层的逻辑
writable._write = function (data, enc, next) {

  // 将流中的数据写入底层

  process.stdout.write(data.toString().toUpperCase())

  // 写入完成时，调用`next()`方法通知流传入下一个数据

  process.nextTick(next)

}

// 所有数据均已写入底层 writable.on('finish', () =>
process.stdout.write('DONE'))
```

```
// 将数据写入流中 writable.write('a' + '\n')writable.write('b' +
'\n')writable.write('c' + '\n')

// 再无数据写入流时，需要调用`end`方法 writable.end()
```

- 上游通过调用 writable.write(data)将数据写入可写流中。write()方法会调用_write()将 data 写入底层。
- 在_write 中，当数据成功写入底层后，**必须**调用 next(err)告诉流开始处理下一个数据。
- next 的调用既可以是同步的，也可以是异步的。
- 上游**必须**调用 writable.end(data)来结束可写流，data 是可选的。此后，不能再调用 write 新增数据。
- 在 end 方法调用后，当所有底层的写操作均完成时，会触发 finish 事件。

Duplex

创建可读可写流。

Duplex 实际上就是继承了 Readable 和 Writable。 所以，一个 Duplex 对象既可当成可读流来使用（需要实现_read 方法），也可当成可写流来使用（需要实现_write 方法）。


```
var Duplex = require('stream').Duplex

var duplex = Duplex()

// 可读端底层读取逻辑 duplex._read = function () {

  this._readNum = this._readNum || 0

  if (this._readNum > 1) {

    this.push(null)

  } else {

    this.push('' + (this._readNum++))

  }

}

// 可写端底层写逻辑 duplex._write = function (buf, enc, next) {

  // a, b
```

```
  process.stdout.write('_write ' + buf.toString() + '\n')

  next()

}

// 0, 1duplex.on('data', data => console.log('ondata', data.toString()))

duplex.write('a')duplex.write('b')

duplex.end()
```

上面的代码中实现了_read 方法，所以可以监听 data 事件来消耗 Duplex 产生的数据。 同时，又实现了_write 方法，可作为下游去消耗数据。

因为它既可读又可写，所以称它有两端：可写端和可读端。 可写端的接口与 Writable 一致，作为下游来使用；可读端的接口与 Readable 一致，作为上游来使用。

Transform

在上面的例子中，可读流中的数据（0, 1）与可写流中的数据（'a', 'b'）是隔离开的，但在 Transform 中可写端写入的数据经变换后会自动添加到可读端。Tranform 继承自 Duplex，并已经实现了_read 和_write 方法，同时要求用户实现一个_transform 方法。

```
'use strict'

const Transform = require('stream').Transform

class Rotate extends Transform {

  constructor(n) {

    super()

    // 将字母旋转`n`个位置

    this.offset = (n || 13) % 26

  }


  // 将可写端写入的数据变换后添加到可读端
```

```
  _transform(buf, enc, next) {
    var res = buf.toString().split('').map(c => {
      var code = c.charCodeAt(0)
      if (c >= 'a' && c <= 'z') {
        code += this.offset
        if (code > 'z'.charCodeAt(0)) {
          code -= 26
        }
      } else if (c >= 'A' && c <= 'Z') {
        code += this.offset
        if (code > 'Z'.charCodeAt(0)) {
          code -= 26
        }
      }
      return String.fromCharCode(code)
    }).join('')

    // 调用 push 方法将变换后的数据添加到可读端
    this.push(res)
    // 调用 next 方法准备处理下一个
    next()
  }

}
```

```
var transform = new Rotate(3)transform.on('data', data =>
process.stdout.write(data))transform.write('hello,
')transform.write('world!')transform.end()

// khoor, zruog!
```

objectMode

前面几节的例子中，经常看到调用 data.toString()。这个 toString()的调用是必须的吗？ 本节介绍完如何控制流中的数据类型后，自然就有了答案。

在 shell 中，用管道（|）连接上下游。上游输出的是文本流（标准输出流），下游输入的也是文本流（标准输入流）。在本文介绍的流中，默认也是如此。

对于可读流来说，push(data)时，data 只能是 String 或 Buffer 类型，而消耗时 data 事件输出的数据都是 Buffer 类型。对于可写流来说，write(data)时，data 只能是 String 或 Buffer 类型，_write(data)调用时传进来的 data 都是 Buffer 类型。

也就是说，流中的数据默认情况下都是 Buffer 类型。产生的数据一放入流中，便转成 Buffer 被消耗；写入的数据在传给底层写逻辑时，也被转成 Buffer 类型。

但每个构造函数都接收一个配置对象，有一个 objectMode 的选项，一旦设置为 true，就能出现“种瓜得瓜，种豆得豆”的效果。

Readable 未设置 objectMode 时：

```
const Readable = require('stream').Readable

const readable = Readable()

readable.push('a')readable.push('b')readable.push(null)

readable.on('data', data => console.log(data))
```

输出：

```
<Buffer 61>

<Buffer 62>
```

Readable 设置 objectMode 后：

```
const Readable = require('stream').Readable

const readable = Readable({ objectMode: true })

readable.push('a')readable.push('b')readable.push({})readable.push(null)

readable.on('data', data => console.log(data))
```

输出：

```
a

b

{}
```

可见，设置 objectMode 后，push(data)的数据被原样的输出了。此时，可以生产任意类型的数据。